SRAM-DRAM Hybrid Memory with Applications to Efficient Register Files in Fine-Grained Multi-Threading

Wing-kei S. Yu, Ruirui Huang, Sarah Q. Xu, Sung-En Wang, Edwin Kan, and G. Edward Suh
Cornell University
Ithaca, NY 14853, USA
{wsy5,rh335,qx33,sw628,eck5,gs272}@cornell.edu

ABSTRACT

Large register files are common in highly multi-threaded architectures such as GPUs. This paper presents a hybrid memory design that tightly integrates embedded DRAM into SRAM cells with a main application to reducing area and power consumption of multi-threaded register files. In the hybrid memory, each SRAM cell is augmented with multiple DRAM cells so that multiple bits can be stored in each cell. This configuration results in significant area and energy savings compared to the SRAM array with the same capacity due to compact DRAM cells. On other hand, the hybrid memory requires explicit data movements in order to access DRAM contexts. In order to minimize context switching impact, we introduce write-back buffers, background context switching, and context-aware thread scheduling, to the processor pipeline and the scheduler. Circuit and architecture simulations of GPU benchmarks suites show significant savings in register file area (38%) and energy (68%) over the traditional SRAM implementation, with minimal (1.4%) performance loss.

Categories and Subject Descriptors

B.3.1 [**Semiconductor Memories**]: Dynamic memory (DRAM), Static memory (SRAM); C.1.0 [**General**]:

General Terms

Design, Performance

Keywords

Memory, Hybrid memory, SRAM, DRAM, GPU, GPGPU, Register file, Fine-grain multithreading

1. INTRODUCTION

As more mainstream applications such as web searches and multimedia processing contain explicit thread-level or data-level parallelism, fine-grained multi-threading has become a popular architecture style to handle long latency operations such as memory accesses without complex dynamic scheduling hardware. For example, today's Graphics Processing Units (GPUs) use a highly multi-threaded architecture where thousands of threads map to each shader processor [7, 6]. Similarly, SUN's T1 and T2 processors contain simple multi-threaded cores to achieve high throughput with low power consumption [9, 11].

To hide the long latency of memory accesses, a fine-grained multi-threading architecture needs to able to quickly switch among a large number of independent threads. For example, an NVIDIA Fermi stream multiprocessor is reported to support 1,536 simultaneously active threads [6]. As a result, register files on such a multi-threaded architecture are much larger than those in traditional processors. The total register files are reported to be 2 MB in an NVIDIA Fermi GPU [6] and 6 MB in AMD Cayman [8].

In this paper, we propose a hybrid memory array that tightly integrates embedded DRAM into SRAM cells and study its use in GPU register files as a main application. In our hybrid memory, each SRAM cell is augmented with N DRAM branches ($2N$ 1T1C DRAM cells) in a way that a value can be locally copied between the SRAM cell and one of the DRAM branches within a cell. The memory allows external access to the SRAM cell, but not directly to DRAM branches. A register file with N contexts can be implemented efficiently with this hybrid array by storing one active context in the SRAM cell and $N-1$ others in DRAM branches. Registers in the active context can be accessed externally. To access others, a dormant context must be explicitly made "active" by copying from DRAM to SRAM. We call this organization a multi-context register file.

Our study through physical layout and SPICE simulations show that the hybrid cell is far more efficient both in terms of area and energy consumption compared to an SRAM array with N times more cells. For example, the layout indicates that for the same amount of bits stored, a hybrid memory array with 4 DRAM contexts per SRAM cell occupies only 62% of the silicon area compared to an SRAM only array because a DRAM branch is much smaller than a 6-T SRAM cell. Similarly, thanks to the reduced capacitance in word-lines and bit-lines, the hybrid memory consumes much less energy for read and write operations. Also, because accesses from a processing unit only see fast SRAM cells, the hybrid design hides delay of traditional DRAM.

However, the use of the hybrid register file presents new architectural challenges due to the restriction that only a subset of registers can be accessed at a time. First, there may be a context mismatch between two pipeline stages. As

ISCA'11, June 4–8, 2011, San Jose, California, USA.

an example, an instruction decode stage may need to read from context 1 while a write-back stage needs to write to context 2. Unfortunately, reading from and writing to two different contexts are not supported by our multi-context hybrid register file. Second, the use of DRAM structure implies that the state is only kept for a short period of time (a few milliseconds) and may need to be refreshed periodically. Finally, context switches may introduce new pipeline stalls that do not exist for today's register files.

In this paper, we show how the GPU pipeline and scheduler can be changed to efficiently handle the above challenges associated with hybrid multi-context register files. A careful investigation shows that a small buffer per context can solve the problem of a context mismatch between the decode and write-back stages by delaying write-backs until the corresponding context becomes active again. The worst-case size of the buffers can be bounded based on the pipeline depth. Interestingly, the DRAM refresh turns out to be almost free in fine-grained multi-threading architectures because each context is likely to be used by a processing unit within a refresh period anyways even without explicit refresh operations. Finally, our architecture hides a latency to switch a context by banking a register file and modifying the scheduler to preload a context in the background.

We studied the implication of the hybrid multi-context register file on the overall GPU performance and the register file energy consumption, using a cycle-level GPU simulator. Fortunately, we found that the performance degradation is minimal with 2 or 4 context register files. For example, with a 3 cycle context switching latency, the average performance degradation is only 0.5% for 2-context hybrid memory and 1.4% for 4-context memory. On the other hand, the hybrid memory provides significant area and energy savings. The area and energy estimates derived from layout, SPICE simulations, and architecture simulations indicate that the 4-context hybrid register file can be implemented with 38% less area and consume 68% less energy on average compared to the traditional SRAM-based register file.

In this paper, we make two main contributions; one in the area of memory cell designs and one in GPU microarchitecture. While embedded DRAM has been used in many different contexts including L3 caches [20], the DRAM array has mostly been designed as a stand-alone structure. We introduce a novel way to combine DRAM branches into each SRAM cell so that a context switch between them can be performed in a parallel fashion with low overheads (Section 3). In addition to the memory design, we propose a set of architectural techniques and a modified scheduler to address limitations of a multi-context register file (Section 4). These architecture techniques are applicable not only to the proposed SRAM-DRAM hybrid memory, but to other register file implementations that limit access to its content.

The rest of the paper is organized as follows. We discuss related work in Section 2. Section 3 introduces our SRAM-DRAM hybrid memory design and explains how the memory cell operates. Using the hybrid memory, Section 4 discusses how a typical GPU pipeline and a scheduler can be modified to incorporate the register file based on the hybrid memory. Section 5 evaluates both the hybrid memory and the overall GPU architecture through circuit-level and architecture-level simulation studies. Finally, Section 6 concludes the paper.

2. RELATED WORK

Embedded DRAM technologies. Embedded DRAM (eDRAM) technologies that are compatible with logic processes have been extensively studied in recent years, especially with the growing gap between on-chip and off-chip bandwidths and the emergence of multimedia applications. In fact, most of the major foundries offer eDRAM as a part of their standard embedded memory package. For example, our hybrid memory design is based on published numbers for IBM's eDRAM based on trench capacitors [10, 2]. Similarly, UMC offers eDRAM with trench capacitors in 90 and 65nm processes [18]. TSMC uses a Metal-Insulator-Metal (MIM) capacitor in their eDRAM offering, which is available from 0.18μ to 40nm processes, and reports two to three times better density compared to the 6-T SRAM [17]. Our hybrid memory cell design does not depend on a particular eDRAM technology and will be applicable to both trench and MIM capacitors.

Use of embedded DRAM. As eDRAM technologies mature, many recent commercial products utilize an eDRAM array as a high-density on-chip memory. For example, IBM Power4 and Power5 implement their L2 caches with the logic-based eDRAM technology [16, 14]. The Power7 microprocessor also includes an eDRAM L3 cache [20]. IBM also used eDRAM in the network data router [5]. As a mass-market product, Microsoft's Xbox360 uses eDRAM on the main processor die [15]. While the proposed hybrid memory also uses the eDRAM technology, this work focuses on integrating SRAM and DRAM together as a hybrid where as the previous examples simply use eDRAM as a discrete memory array.

SRAM-DRAM hybrid. Recently, Valero et al. proposed a SRAM-DRAM hybrid macrocell that is designed to implement first level data caches [19]. While the high-level approach to combine SRAM and DRAM technologies is similar to our hybrid memory, their cell design is quite different from ours and not efficient for a register file. More specifically, their macrocell stores n-bits using one 6T SRAM cell and $(n-1)$ 1T1C DRAM cells. Both SRAM and DRAM cells are accessible externally and data can only transfer internally from SRAM to DRAM. On the other hand, our hybrid memory can internally move data between SRAM and DRAM while only allowing access through SRAM cells. In addition to the difference in the memory cell designs, this work also investigates architectural mechanisms to enable efficient use of hybrid memory in register files for fine-grained multi-threading.

Multi-context memory. Register files in multi-threaded architectures commonly store registers from multiple threads. In this sense, these register files are also *multi-context*. However, in this paper, we use the term multi-context memory to indicate that memory is partitioned and only one partition (context) can be accessed at a time.

Conceptually, our definition of a multi-context memory is almost identical to the way register windows function in RISC I [13] or SPARC ISA. In fact, the register windows also allow efficient implementations through combining multiple memory technologies. For example, the register file in the UltraSPARC T1 microprocessor is composed of a multi-ported register file backed by a more compact SRAM array [9]. The current register window is accessible via the register file, whereas other register windows are stored in the SRAM until needed. While the high-level idea of combining

different memory structures in register file is similar to register windows, this paper studies hybrid cells that combine SRAM and DRAM rather than two types of SRAMs. Also, context switches in our hybrid memory are quite frequent and managed by hardware whereas typical register windows are managed in software and switched rather infrequently on function calls and returns. We believe that our hybrid memory design is applicable to register windows as well even though DRAM refresh will need to be handled more carefully.

3. MULTI-CONTEXT MEMORY USING SRAM-DRAM HYBRID CELLS

In a traditional memory array, any memory location may be accessed at any given time, often with the same cost. Such a uniform organization often implies that all locations are implemented with a single memory technology. Here, we introduce a notion of a multi-context memory array and show how SRAM and DRAM technologies can be tightly integrated to provide an efficient implementation.

3.1 Multi-Context Memory

We define a multi-context memory as a memory array where cells are partitioned into multiple contexts, out of which only one active context is accessible at a time. To access other contexts, termed dormant contexts, the active context must be explicitly switched. Conceptually, an N-context memory consists of a memory array for one active context and N back-up arrays for dormant contexts.

For accesses to the active context, the multi-context memory behaves identically to a traditional SRAM array. Normal read and write operations are not affected by having multiple contexts except that they can only access the active context. To manage data in the dormant contexts, two new operations to store and load contexts are introduced:

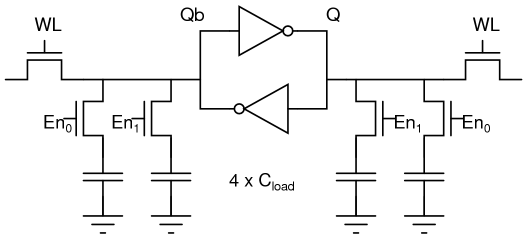
- *Store context i*: Save the active context to a back-up array i. The active context is not affected and remains available.
- *Load context i*: Copy a context from a back-up array i to the memory array for the active context. This operation overwrites data in the active context.

While performing a store or load of a context, the multi-context memory will not be able to service regular read and write operations. The regular read or write will be delayed until the store or load of a context is complete.

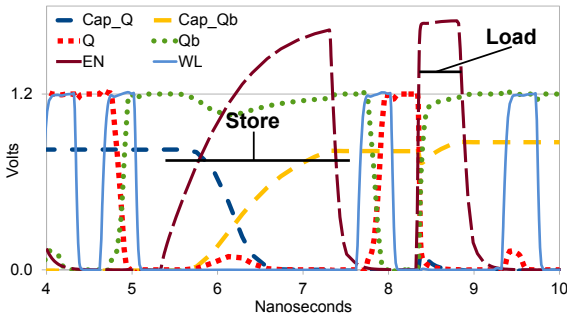
3.2 SRAM-DRAM Hybrid Cell

The multi-context memory, thanks to its explicit management of active and dormant contexts, allows a memory array to potentially be implemented more efficiently as a hybrid of multiple memory technologies. In this paper, we propose to tightly integrate SRAM and DRAM technologies to construct a new hybrid memory cell that is suitable for a multi-context memory array.

To build a N-context SRAM-DRAM hybrid, a typical 6-transistor SRAM cell is augmented with N 1-transistor 1-capacitor DRAM cells at both `Q` and `Qb` nodes. Figure 1(a) shows the circuit design for $N = 2$. More pairs of DRAM cells can be attached to support more dormant contexts.



(a) Hybrid memory cell circuit



(b) Storing and restoring a context

Figure 1: A hybrid SRAM-DRAM cell with an SRAM bit for the active context and DRAM bits for dormant contexts.

The DRAM capacitors are isolated from the SRAM's cross-coupled inverters by DRAM enable transistors during normal read/write operations. By controlling the enable transistors, a value can be copied between the SRAM cell and DRAM cells. The circuit uses a pair of DRAM cells to store one dormant context.

We choose an SRAM cell to store the active context as it provides fast access. Dormant contexts are stored in DRAM cells in order to minimize the area and the power consumption, providing extra capacity at minimal cost. As an example, the area of an embedded 1-T DRAM cell is reported to be as small as 15% of an SRAM cell in a 65nm process technology [2].

The cell in the figure is capable of storing 2 contexts (1 bit per context) using the two DRAM branches (4 DRAM cells). The SRAM cell is only used to make a context in DRAM available and cannot be used as extra storage because a read from DRAM overwrites active context in the SRAM cell. As a result, an active context in SRAM must be copied to a DRAM before another context is restored.

Read and write operations for the active context in the hybrid memory cell are conducted in the same manner as with a traditional SRAM cell as the active context is stored in the SRAM cell. During read and write operations, the DRAM enable transistors connecting the SRAM cell and the DRAM capacitors are turned off, isolating the DRAM cells from the SRAM.

A context store operation is performed by writing a bit in the SRAM cell into the selected dormant DRAM context as shown in Figure 1(b). The values on the `Q` and `Qb` nodes are copied into a pair of DRAM capacitors, one connected to `Q`

and the other connected to Qb. Loading a dormant context into the SRAM is done by reading from the selected pair of DRAM capacitors. The data in the SRAM context would be lost unless first saved.

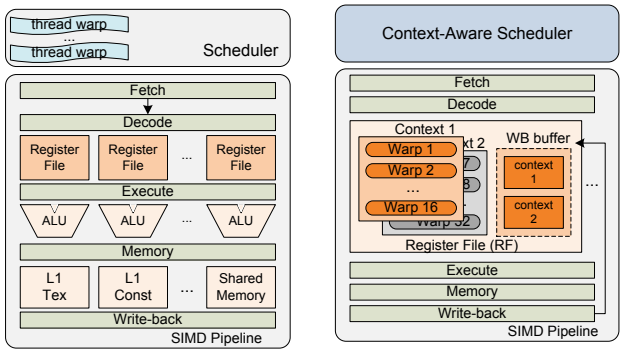
Both context store and load operations are performed by turning on the DRAM enable transistors, connecting Q and Qb nodes to the corresponding DRAM capacitors. At a circuit level, we design the DRAM capacitance to be larger than the Q and Qb node capacitance, and the load and store operations are differentiated by the ramp time of the enable signal for transistors connecting DRAM capacitors to SRAM nodes. A fast ramp time (< 0.1 ns in our circuit) connects the larger DRAM capacitance to the SRAM state node abruptly and the SRAM node swings toward the DRAM capacitance (load a context from DRAM). A slower ramp time allows the SRAM state node enough time to drain or fill the DRAM capacitor to its own value (store a context to DRAM). Wordlines and bitlines stay off. The waveforms in Figure 1(b) depict storing a 0 to the DRAM, writing a 1 to the SRAM, then loading the stored 0 from the DRAM.

The functionality of the hybrid cell has been studied via HSPICE simulations and a layout of a small array in IBM 65nm process technology. The SPICE simulations validated the correctness of the operations. The retention time of a DRAM content is mainly determined by the size of the DRAM capacitor and the length of the DRAM enable transistor, which affects the leakage current. In our circuit design, simulations suggest that a retention time of around 2.3 ms can be achieved with a capacitor size of 7.5 fF. We also studied the effects of additional capacitance, the timing of enable signals, the impact of process variations, etc., to further verify the hybrid memory functions. Section 5.2 provides detailed evaluations of the area, energy consumption, and delay of the hybrid memory array compared to traditional SRAM arrays.

3.3 Strengths and Weaknesses

Utilizing the efficiency of DRAM, the SRAM-DRAM hybrid memory can provide the same storage capacity with significantly less silicon area and energy consumption, compared to a traditional SRAM array. One SRAM-DRAM hybrid cell with 2 DRAM contexts can store the same amount of data as two SRAM cells. The layouts suggest that a 2-context hybrid memory cell occupies 1.8 times the area of a single SRAM cell and a 4-context hybrid cell consumes only 2.4 times the area of a single SRAM cell. As a result, the 4-context cell reduces the area almost by half compared to 4 SRAM cells. The area reduction leads to lower leakage power and also lowers dynamic power consumption for read and write operations because bitlines and wordlines for the memory array become shorter.

In exchange for the efficiency, the hybrid memory trades-off data accessibility. The hybrid memory does not allow accesses to data in a dormant context unless there is an explicit context switch. On the other hand, in a traditional SRAM array, any location can be accessed without any restriction. The context switches cost additional latencies and energy consumption, especially if contexts need to change frequently. The use of DRAM cells also imply that explicit refresh operations are required in order to keep state in dormant contexts for an extended period because the charges in the DRAM capacitors will eventually leak out.



(a) Baseline (b) Multi-context RF

Figure 2: GPU processing engine block diagrams with single-context and multi-context register files.

4. GPU ARCHITECTURE WITH MULTI-CONTEXT REGISTER FILE

In order to effectively apply the hybrid memory as register files in fine-grained multi-threading architectures such as GPUs, the architecture needs to be able to properly compensate or hide the weaknesses of the hybrid memory, namely limited accessibility and overheads for context switches and DRAM refreshes. In this section, we discuss how a GPU pipeline and a thread scheduler can be changed to utilize the hybrid memory with minimal impact on performance.

4.1 Baseline GPU Pipeline

We use a typical General-Purpose GPU (GPGPU) processing pipeline as the baseline for our discussions. Computation in a GPGPU occurs in a highly parallel manner. A GPGPU consists of many cores, often called stream multiprocessors (SM), each of which again contains multiple execution units. The GPGPU pipeline often issues threads in groups of 32, called a warp. Warps are issued atomically and all threads in a warp must be ready to execute in order for a warp to be issued. Threads in a warp get executed in parallel by multiple execution units in a core. As an example, NVIDIA GT200 with 8 execution units takes 4 pipeline cycles to execute a warp with 32 threads [7].

To fully utilize the pipeline and hide a long memory access latency, each GPGPU core supports thousands of threads. A hardware scheduler operating on warp granularity determines which warps are ready and selects one to issue. Typically, there is no penalty in switching from one warp to another. Therefore, even if some warps are stalled for long latency operations, the overall pipeline can be kept busy with other warps.

For this fine-grained multi-threading scheme, each thread needs its own set of architectural registers in the core register file. As a result, the register file in a GPGPU core is large. Recent NVIDIA GPGPU register files have 32,768 entries of 32 bits each, resulting in 128KB per core and 2MB for a chip [6]. As a comparison, typical high-performance general purpose processors have much smaller register files in the range of 8 to 16KB per core as they contain low hundreds of entries of 64 bits each.

Figure 2(a) shows the baseline GPU pipeline that uses a typical SRAM-based register file. While commercial GPUs contain specialized blocks for graphics, we only discuss a general processing pipeline in this paper because we are

concerned with a GPU as a general-purpose platform. Our baseline pipeline consists of fetch, decode, execute, memory, and write-back stages that are similar to the traditional processor pipeline. This view of the GPGPU pipeline has been borrowed from the previous work on GPU modeling [1] based on NVIDIA documents [12]. Register file accesses occur in the decode and write-back stages.

GPGPU cores typically contain multiple execution units that operate in parallel. In this sense, as shown in the figure, the GPGPU pipeline has a collection of multiple parallel pipelines, one for each execution unit. In the following discussions, we will refer to each execution pipeline as a *lane*. The multiple lanes in a core imply that the GPU register file needs to support many reads and writes in each cycle. For example, the NVIDIA GT200 GPU has 8 lanes per core where each instruction may read up to 3 registers and write one register. This implies up to 24 reads and 8 writes per cycle. Because highly-ported register files are expensive, a typical GPU relies on banked register files where each lane reads from its own register file bank. This banked design implicitly restricts each thread to only be able to execute on a particular lane. For example, the 1,024 threads in a GT200 core are statically partitioned into 8 lanes. As a result, one register file bank only needs to contain registers for 128 threads.

4.2 Pipeline with Multi-Context Register Files

Figure 2(b) shows changes to the baseline GPU pipeline when a multi-context memory register file is used. As in the baseline case, the register file is banked so that each bank only needs to support one lane. However, each bank will use hybrid memory cells to reduce its area and power overheads. As shown in the figure, with a multi-context register file, only a subset of the registers in an active context are accessible from the pipeline. Registers in the other contexts are stored in dormant arrays and need to be explicitly switched into the active context to be accessible. For example, a 16-KB SRAM bank for a traditional register file can be replaced with a 4-context memory array with 4-KB contexts (4-KB SRAM backed by 4 4-KB DRAM contexts) or a 2-context array with 8-KB contexts (8-KB SRAM baked by 2 8-KB DRAM contexts), etc.

In multi-context register files, the registers must be divided into each context. In a GPGPU, a logical granularity for this assignment would be a warp, as threads from a warp are scheduled together, implying that their registers will be accessed together. To minimize the number of unnecessary context switches in the register file, the context assignment should keep threads from a warp in the same context and minimize the number of overall contexts. We achieve this objective by sequentially assigning each warp into each context in order. For example, if only the first 16 out of 32 warps are active, and there are four contexts for a register file, warp 1 to 8 will be assigned to the first context, and warps 9 to 16 will be assigned to the second context, using only 2 of the 4 contexts.

While a multi-context register file can bring significant advantages in area and power consumption, its limitations also pose new challenges in the architecture design. The following discussions address how the three major challenges, namely context mismatches among pipeline stages, context switch overheads, and DRAM refresh overheads, can be addressed in our architecture.

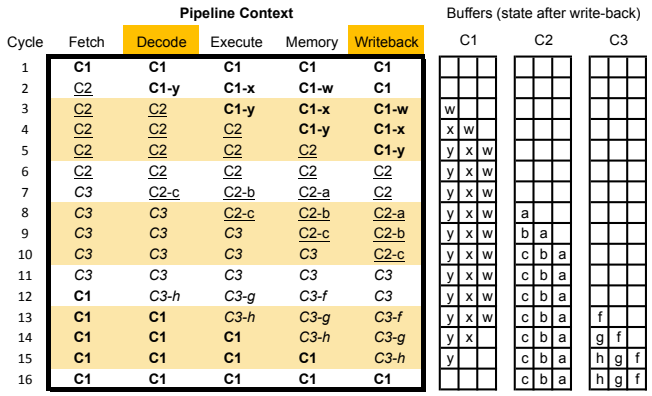


Figure 3: Context mismatch between pipeline stages and our solution with context write-back buffers.

4.2.1 Context Mismatch between Pipeline Stages

The multi-context register file implies that a processing core may switch an active context depending on which warp is being issued. Therefore, different stages of the pipeline may need data from different contexts simultaneously. For example, a register in context 1 may be used in the write-back stage, while a register in context 2 needs to be read as an operand in the decode stage. The accesses to the two different contexts pose a conflict because the multi-context register file can only allow accesses to one context at a time. Figure 3 illustrates an example of such a conflict with three contexts: C1, C2, and C3. In cycle 3, C2 needs to read registers while C1 needs to write a result back.

A naive way to avoid such a structural hazard is to complete all pipeline stages for instructions from one context before switching register contexts and issuing instructions from a new context. Essentially, this method stalls the decode stage until preceding instructions complete on every context switch. As a result, this approach can add a large number of stalls and significantly degrade performance.

To avoid stalls, a warp from a new context must be able to continue its execution without waiting for processing of warps from the previous context to be completed. Unfortunately, this optimization implies that the write-back stage of the pipeline cannot write results back because the register file is already switched to another context. To address this problem, we add a small write-back buffer for each context to the register file. On a context switch, a write-back stage can put its result into the buffer for its context while the decode stage can read from the new context. Later, the entries from the write-back buffer can be put into the register file when the corresponding context becomes active again.

Figure 3 shows operations of the multi-context register file with per-context write-back buffers shown on the right side. After the first context switch from C1 to C2 in cycle 3, results from the instructions belonging to C1, tagged with *w,x,y*, are written to C1's write buffer. When the context is switched back to C1 in cycle 13, actual C1 write-backs to the register file take place while C3 writes to its buffer. Note that there will be no register port contention for such delayed write-backs. After a context switch, the write-back stage in the pipeline will write to the buffer for the previous context, while the write-back buffer for the current context empties out to the register file. In effect, this scheme delays a particular context's write-back to the next time it is switched

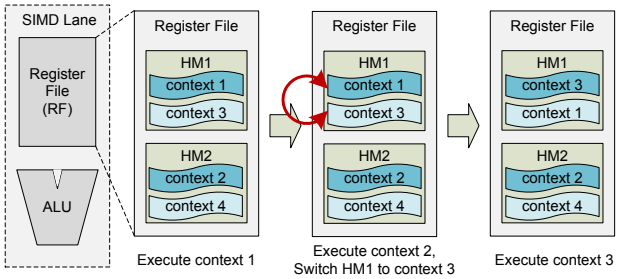


Figure 4: A background context switch using two register sub-banks per lane.

into the active context. The size of each write-back buffer can be bounded by the number of pipeline stages between the decode and the write-back.

Note that on a read, with this optimization, the register file now needs to check the current context's write-back buffer. If there is a match, the value from the write-back buffer needs to be used instead in the pipeline. Because the write-back buffers are small, with only several entries to tentatively delay write-backs, a look-up to the write-back buffer can be performed in parallel to a register file read without any performance impact.

4.2.2 Context Switch Latencies

With the multi-context register file, the pipeline can only access registers from an active context. Once all warps in the active context become blocked, the core needs to explicitly make one of the dormant contexts active before it can issue warps from other contexts. Unfortunately, these register context switches can stall the decode stage of the pipeline because the register file becomes inaccessible during the switch. To alleviate the context switch overheads, as discussed before, we carefully place warps into contexts in a way that minimizes the number of active contexts given the number of active warps. This warp placement aims to reduce the number of context switches. However, because context switches may still happen rather frequently, we also need a way to hide the context switching latency without stalling the pipeline.

To hide the context switch latency, we propose to use two sub-banks for a register file bank in each lane so that a context switch of one sub-bank can be performed in the background while the other sub-bank is being used by the pipeline. Figure 4 illustrates how banking enables background context switches. As shown in the figure, one hybrid memory bank is split into two sub-banks, each with half the size of the original one. For example, a 16-KB 2-context memory array (8-KB SRAM with 2 8-KB DRAM) is replaced with two 8-KB 2-context arrays (4-KB SRAM with 2 4-KB DRAM). In the following discussion, we call the two sub-banks as Hybrid Memory 1 (HM1) and Hybrid Memory 2 (HM2). HM1 contains context 1 and 3, and HM2 contains context 2 and 4.

The pipeline starts issuing warps from an active context of one of the sub-banks, say context 1 from HM1. When HM1 needs to switch its context, the pipeline can issue warps from the active context of the other sub-bank, say context 2 from HM2. While executing warps from HM2, HM1 can switch from context 1 to context 3 in the background. By the time that HM2 needs to switch its context, the context switch for HM1 would be completed so that warps can be issued from HM1 without stalling the pipeline. If warps from HM2 are blocked before the context switch is done, the switching latency is only partially hidden. Section 4.3 discusses the warp scheduling algorithm, which controls context switches, in more detail.

4.2.3 DRAM Refresh

Our hybrid memory cell uses DRAM capacitors as storage elements for dormant contexts. Because DRAM leaks charge over time, this construction implies that dormant contexts need to be periodically refreshed - loaded to the active context and stored back. For example, our reference design has a retention time of 2.3 ms. A longer time could be achieved by increasing the DRAM capacitor size, at the cost of more energy expended during store operations.

To ensure that each dormant context is refreshed within the retention time, we add a refresh timer for each context to keep track of the number of cycles since the last time the context has been written to DRAM. If a refresh timer is close to the retention limit, the schedule is forced to bring the corresponding context to SRAM as an active context. Obviously, this operation also requires the current active state to be copied into DRAM. This refresh operation may incur pipeline stalls if there is no warp to issue from another sub-bank.

In practice, however, our experiments indicate that fine-grained multi-threading architectures tend to re-schedule each warp within a reasonably short period. As a result, each dormant context almost always gets brought back to SRAM as an active context before its refresh timer reaches a retention limit. As a result, we found that the DRAM refresh typically does not interrupt normal pipeline operations.

4.3 Context-Aware Warp Scheduling

In our architecture, a scheduler needs to perform two major functions. First, similar to the baseline GPU, the scheduler needs to select which warp will be issued in each cycle. However, unlike the baseline, the scheduler is restricted to choose a warp from the active contexts of two register file sub-banks. Additionally, the scheduler needs to also manage multiple contexts for each sub-bank by deciding when a context switch should happen and which context should become active. The context switches need to be carefully managed considering multiple factors including DRAM refreshes, context switch overheads, and fairness.

In this subsection we discuss how the warp scheduling algorithm can be augmented to accommodate additional constraints from the hybrid memory. More specifically, there are three major design considerations for the scheduler:

- *Correctness*: The scheduler must ensure that dormant contexts in DRAM are preserved by reloading each context before it reaches the DRAM retention limit.
- *Efficiency*: To minimize pipeline stalls and context switch energy, the scheduler needs to minimize unnecessary context switches and intelligently schedule context switches to hide their latencies.
- *Fairness*: The scheduler needs to be fair to all warps, providing equal opportunities to execute. Unfair scheduling can lead to a noticeable performance degradation by leaving only a small number of warps towards the end of program execution.

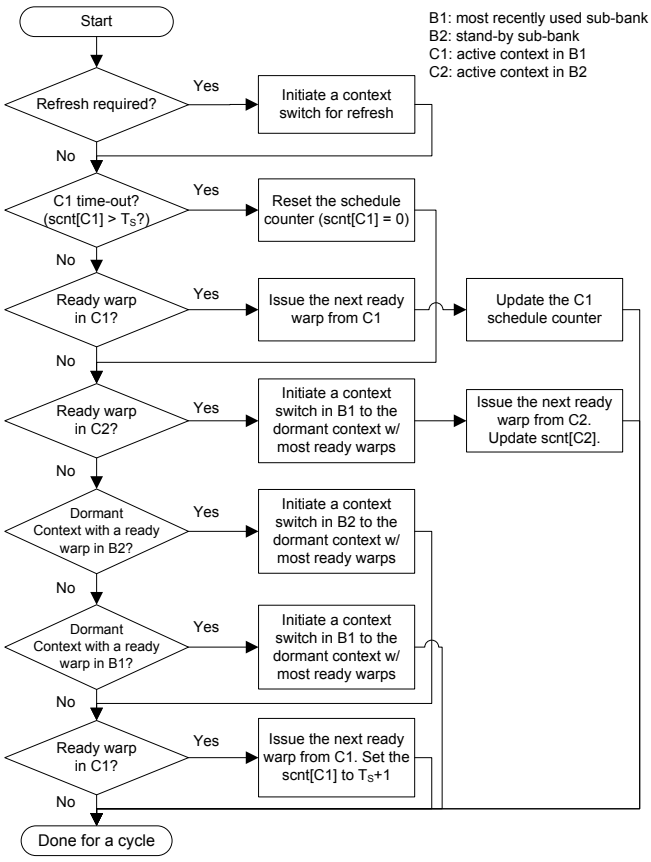


Figure 5: Flow chart for warp scheduling and register file context switching.

While it is relatively straightforward to deal with the DRAM refresh requirement, we found that there is a fundamental tension between reducing the number of context switches and improving fairness. In order to minimize context switches, the scheduler should issue warps from one active context as long as there is a "ready warp" (warp that is ready to be issued) in the context. However, such a greedy algorithm may lead to unfairness, especially when a subset of contexts have more active warps than others; contexts with more active warps are more likely to get chances to execute. We found that the unfairness can significantly degrade performance for some applications by leaving a small number of warps to run towards the end of an execution without enough warps to fully utilize the pipeline.

Our scheduling algorithm deals with the tension between context switch overheads and fairness by setting a threshold on how long that each context can run without switching to another context with ready warps. While this threshold does not strictly guarantee fairness, we found that the scheme works quite well in practice, close to more complex scheduling algorithms with stronger fairness guarantees. The evaluation section provides further discussions on the trade-off between context switches and fairness based on simulations.

Figure 5 shows the flowchart for the new scheduler. To illustrate how a scheduling algorithm is changed for the multi-context register files, we use a simple baseline scheduling algorithm, which selects a ready warp in a round-robin fashion. To aid its decisions, the scheduler maintains three types of counters per context: refresh counter ($rcnt$), schedule counter ($scnt$), and ready-warp counter ($wcnt$). The refresh counter keeps track of when a DRAM context needs to be refreshed. The schedule counter indicates how many times a warp in a context has been checked for possible issue by the round-robin scheduler without a forced context switch. The ready-warp counter indicates how many warps are ready for each context.

As shown in the figure, in each cycle, the scheduler first checks if any dormant context needs to be refreshed, and initiates a context switch to make the dormant context active if necessary. The corresponding sub-bank is excluded from scheduling decisions until the switch is completed.

To determine which warp to issue in a given cycle, the scheduler first tries to select a ready warp from the active context (C1) of the most recently used sub-bank (B1) in a round-robin fashion. If the schedule counter for C1 ($scnt[C1]$) reaches a fairness threshold (T_S) or there is no ready warp in C1, the scheduler tries to pick a ready warp from the active context (C2) of the other sub-bank (B2, called stand-by sub-bank) and initiates a background context switch in B1 if successful. If there is no ready warp in the stand-by sub-bank (B2), the scheduler searches for a dormant context with a ready warp and initiates a context switch. To reduce future context switches, the scheduler selects a dormant context with the largest number of ready warps on a context switch. Finally, if no warp from another context is ready, a ready warp from the current active context (C1) is allowed to issue even if the threshold is reached.

While not shown in the flowchart to keep the chart simple, our scheduler implementation contains one more optimization to hide context switch latencies. In each cycle, while issuing a warp from one sub-bank, the scheduler checks how many ready warps are left in the current active context of that sub-bank. If the number is low, just enough to hide a context switch latency, and there is no ready warp in the active context of the stand-by sub-bank, the scheduler initiates a context switch in the stand-by sub-bank.

5. EVALUATION

This section evaluates the performance, area, and energy consumption of the proposed hybrid memory and GPU architecture through detailed simulations.

5.1 Experimental Methodology

To validate functionality and study performance of the hybrid memory, we used SPICE simulations along with cell layout. The SRAM-DRAM hybrid cell was simulated in HSPICE using IBM 65nm process technology transistor models. The cell was laid out using custom layout with IBM 65nm design rules. For the embedded DRAM, the area and placement are estimated based on the previously reported numbers from an eDRAM array [2]. The hybrid memory arrays were laid out in sizes from 16x1 to 16x64, extracted and simulated to obtain area and energy measurements. Array area and energies for larger arrays are estimated by scaling the bitline and wordline capacitances, which are predictable from the array size. We also checked this scaling method by designing several memories with a commercial memory compiler.

In this study, we use single-port SRAM arrays rather than multi-ported ones to evaluate our hybrid memory design. We believe that GPU register files are often heavily

Parameter	*Value*
Shader cores	30
Threads per core	**1024**, 2048
Threads per warp	32
Warps per core	**32**, 64
Registers per core	**16384**, 32768
Register file size per core	**64KB**, 128KB
Register file banks per core	32
Register file bank size	**2KB**, 4KB
Register file sub-banks	2
Execution units per core	16, **32**
Hybrid Memory Register File	
Contexts per cell	2, 4, 8
Context switch latency	**3**, 4, 5
Scheduler counter threshold	**10,000**

Table 1: GPGPU-Sim parameters and hybrid memory configurations. Bold denotes default values.

banked and use single-port SRAM arrays to save silicon area. The single-port configuration also provides conservative estimates for hybrid memory overheads. In multi-ported arrays, a part of the DRAM branch overhead can likely be hidden by additional bit-lines.

For GPGPU performance estimates and counting events for energy estimates, we modified GPGPU-Sim v2.1.1.b [1] to model the new scheduler and the hybrid register file. Simulator parameters are shown in Table 1. In the default configuration, each shader core has 32 execution units and includes a 64-KB register file. As a result, the core can complete one warp per cycle. The register file is assumed to be split into 32 banks of 2KB each, one per execution unit. Each bank is also assumed to have two 1-KB memory arrays (sub-banks), which can be used for background context switches as discussed in Section 4.2.2. For fair comparisons, both the baseline SRAM register files and the hybrid register files use the same sub-bank configuration.

For the hybrid register file, we studied 2, 4, and 8-context configurations to implement each sub-bank. For example, a 1-KB sub-bank array can be implemented with a 1-KB SRAM array, a 2-context hybrid array with 512 bytes (8 warps) per context, a 4-context hybrid array with 256 bytes (4 warps) per context, or 8-context hybrid array with 128 bytes (2 warps) per context. To study the impact of various architecture parameters on the hybrid register file performance, we tried different context switch latencies (3, 4, and 5), a narrower pipeline (16 execution units), and more threads per core (2048 threads with 128-KB register file).

The energy consumption of a register file is estimated by combining results from the circuit and architecture simulations. The read, write, and context switch energies for a 1-KB sub-bank are obtained from SPICE simulations. Then, the overall energy consumption is computed by multiplying the per-event energy with the number of events that are counted by the architecture simulator.

Table 2 shows the benchmarks that are used in the simulations and their characteristics. The benchmarks were taken from the GPGPU-Sim benchmark suite [1] and the Rodinia GPGPU benchmark suite [3, 4]. They comprise compute-heavy applications meant to be run in the massively parallel environment of a GPU, and cover a wide range of register file usage. Some benchmarks require a small number of registers while others heavily stress the register file.

5.2 Hybrid Memory

For the hybrid SRAM-DRAM memory cell, we studied configurations with 2, 4, and 8 DRAM contexts attached to each SRAM cell. We refer to these designs as 2-context, 4-context, and 8-context configurations.

Name	*Description*	*Warps/ core*	*Inst. count*	*Baseline IPC*
LPS	3D Laplace Solver	16	82M	551
BLK	Black-Scholes option pricing	24	196M	804
NQU	N-Queens Solver	3	1.3M	42.4
MUM	MUMmerGPU	32	75M	62.2
RAY	Ray Tracing	12	65M	722
STO	StoreGPU	4	124M	734
WP	Weather Prediction	6	217M	215
BFS	Breadth First Search	32	17M	48.7
NN	Neural Network Digit Recognition	30	68M	29.2
R_BFS	Breadth First Search	32	484M	97.4
R_HS	Thermal Simulation	16	80M	830
R_LUD	LU Decomposition	32	40M	63.5
R_NW	Needleman-Wunsch DNA alignment	5	218M	48.7

Table 2: Benchmark characteristics. Last 4 benchmarks from Rodinia suite. Warps/core: maximum number of warps assigned at a time per core.

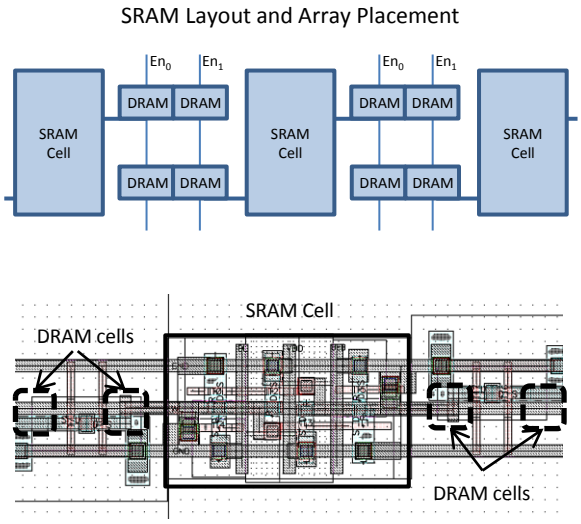


Figure 6: SRAM with 2 contexts layout.

5.2.1 Cell-level Discussion

Figure 6 shows our layout for the 2-context hybrid SRAM-DRAM cell. The block diagram in the top half of the figure shows the floorplan of an array. The DRAM cell 'leaves' from adjacent SRAM cells would overlap and share space, allowing for a compact design. The image in the bottom half of the figure is a screen shot of the actual layout for the 2-context SRAM-DRAM cell, with the SRAM and DRAM cells highlighted.

Area estimates from extracted layout of an SRAM and

Parameter	*SRAM*	*Hybrid 2-context*	*Hybrid 4-context*	*Hybrid 8-context*
Width (μm)	1.84	3.37	4.47	6.67
Height (μm)	0.7	0.7	0.7	0.7
Area (μm^2)	1.29	2.36	3.13	4.67
Relative sizes	100	183.2	242.9	363.5

Table 3: SRAM-DRAM hybrid memory cell area comparison.

Bytes Stored					
-	128	256	512	1024	2048
SRAM array energy cost					
Cells in array	1024	2048	4096	8192	16384
Read energy (fJ)	919	1830	3660	7320	14600
Write energy (fJ)	757	1510	3030	6060	12100
Leakage energy (nW)	202	400	795	1590	3170
Hybrid SRAM-DRAM, 2 context energy cost					
Cells in array	512	1024	2048	4096	8192
Read energy (fJ)	526	1040	2070	4140	8270
Write energy (fJ)	502	944	1830	3600	7140
Context switch (pJ)	5.12	10.2	20.5	40.9	81.9
Leakage energy (nW)	149	296	588	1170	2340
Hybrid SRAM-DRAM, 4 context energy cost					
Cells in array	256	512	1024	2048	4096
Read energy (fJ)	287	535	1030	2030	4020
Write energy (fJ)	227	456	914	1830	3660
Context switch (pJ)	3.69	7.38	14.8	29.5	59.0
Leakage energy (nW)	71.0	142	284	568	1140
Hybrid SRAM-DRAM, 8 context energy cost					
Cells in array	128	256	512	1024	2048
Read energy (fJ)	178	303	553	1050	2050
Write energy (fJ)	116	239	485	977	1960
Context switch (pJ)	2.01	4.03	8.05	16.1	32.2
Leakage energy (nW)	39.4	76.0	149	295	588
Area in μm^2					
SRAM	1529	3059	6118	12235	24471
2 context SRAM-DRAM	1471	2871	5673	11275	22480
4 context SRAM-DRAM	1057	1985	3843	7559	14990
8 context SRAM-DRAM	935	1628	3014	5786	11330

Table 4: SRAM-DRAM hybrid memory array energy and area estimates.

hybrid SRAM-DRAM cell appear in Table 3. Adding 2 contexts to the SRAM cells costs another 83% area overhead compared to one SRAM cell. With 4 contexts, we note that the area overhead is only 140% more than a single SRAM cell. Compared to 4 SRAM cells holding the same amount of data, the area of a hybrid SRAM-DRAM cell with 4 contexts is 39% smaller.

At the cell level, the additional DRAM branches can have noticeable impact on memory latencies. Simulations indicate that the read delay at the cell level is increased by 70% in the 8-context hybrid cell. The write delay increases by 116%. The 4-context hybrid cell read and write delays are increased by 38% and 64%, respectively.

5.2.2 Array-Level Discussion

In memory arrays, the read/write latency and energy are dominated by the bitline and wordline capacitances. As a result, the array-level characteristics are largely determined by the size of an array, and the additional DRAM capacitance at each SRAM cell only has a minor impact. Table 4 shows the array-level energy and area estimates for different array sizes. Because the hybrid memory array is physically much smaller and requires fewer SRAM cells than the corresponding SRAM array of equal data storage, its bitline capacitance is also smaller, which results in much lower energy consumption for read and write operations. For example, we find that a 1-KB SRAM array consumes 7.32pJ for a read whereas a 1-KB (2048 cells) 4-context hybrid array only consumes 2.03pJ per read operation. This represents an reduction in read energy of 72%. The hybrid memory also shows a similar reduction in write energy. The table also shows that the energy savings for regular read/write operations are more significant with more DRAM contexts thanks to further reduction in area and memory cells.

At a circuit-level, there is a large design space for a memory array. For example, peripheral circuits can be sized differently in order to provide a different trade-off point between delay and energy. In this study, given that we use the hybrid memory to replace an SRAM-based register file in a pipeline, we sized read and write circuits so that the read/write latency of all SRAM and hybrid memory arrays are roughly identical. For example, a hybrid memory array with additional DRAM capacitance uses larger read/write drivers compared to the SRAM array with the same number of cells. Larger arrays also use larger read/write circuits. The read and write latencies for our arrays are 130ps and 140ps, respectively. We also note that the read and write circuits could be further optimized.

In addition to regular read and write operations, the hybrid SRAM-DRAM memory needs to move data between SRAM and DRAM. Table 4 shows the estimated context switch energy for different array sizes, which includes energy consumption within each cell as well as energy to change control lines (EN). For typical array sizes, the estimated energy consumption of a context switch is comparable to a few tens of read operations. In terms of latency, our SPICE simulations show that the hybrid SRAM-DRAM cell can switch a context (one SRAM-to-DRAM store and one DRAM-to-SRAM load) under 4 ns (see Figure 1(b)). This is a conservative estimate and we believe that the switch can be even faster if necessary. Given that current GPU cores run at around 700 MHz, a context switch will take about 3 core clock cycles. To see the impact of switching latency, we also simulated 4 and 5 clock cycle latencies in later studies.

Table 4 also shows area estimates based on the layout. The estimates only include the area of an actual cell array without any peripheral circuitry. Measurements are shown in μm^2. The 2-context hybrid memory arrays are roughly 7% smaller than SRAM arrays with the same capacity, while the 4-context and 8-context memories use 38% and 52% less area, respectively.

Retention time is another important consideration for a DRAM-based memory. Larger storage capacitors allow a longer retention time, but cost more energy on a context switch and take up more silicon area. For our hybrid SRAM-DRAM cell, we chose a 7.5fF capacitor with 2.3 ms retention time. In terms of GPU core cycles, this retention time presents over 1.4 million cycles.

5.3 GPU Performance

Figure 7 shows the performance of hybrid register files for a different number of contexts. The performance is normalized to the baseline SRAM register file. The figure shows that the performance overhead increases with more DRAM contexts per cell. This trend is expected as a fewer number of contexts implies more warps for each context, which leads to fewer context switches. Even with context switches, the overall performance loss is quite low for the 2 and 4-context configurations: 0.5% and 1.4%, respectively. On the other hand, the 8-context configuration yields high performance overheads in several benchmarks due to a greater number of context switches. The 8-context configuration shows an average of 13.3% performance loss. Note Gmean denotes geometric mean for all figures in this section.

Overall, we found that there are two main sources of performance overheads in hybrid register files: additional context switches and changes in warp scheduling. Most of performance overheads can be explained with the number of

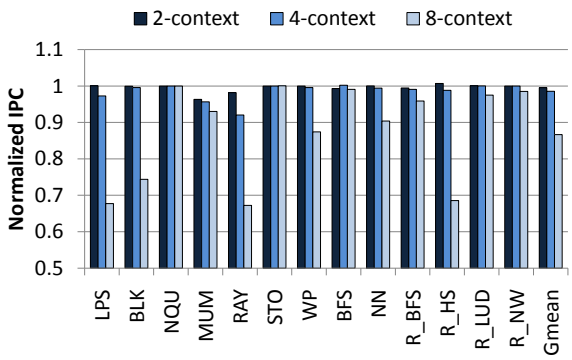


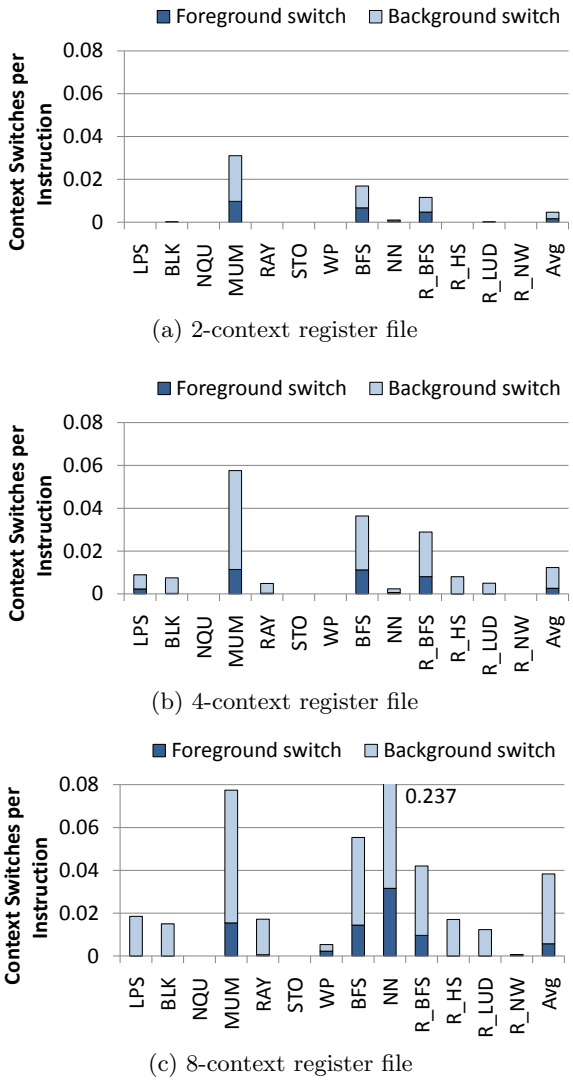
Figure 7: Performance comparisons for hybrid register files (2, 4, and 8 contexts) under 3 cycle context switch latency and 32 execution units per core. The performance is normalized to the baseline SRAM register file.

context switches while some benchmarks suffer from scheduling restrictions imposed by contexts. The following paragraphs discuss these two factors in more detail.

Figure 8 shows the frequency of context switches for the three context configurations. In the figure, we categorize context switches as either background or foreground. The background switches happen in the standby sub-bank while a pipeline continues using the other sub-bank. Therefore, a background switch latency can be hidden either partially or entirely depending on the number of ready warps. On the other hand, the foreground switches happen when there is no ready warp available in another sub-bank. It is important to note that not all visible context switches cause additional pipeline stalls because context switches may overlap with other types of stalls. For example, if a pipeline is stalled at the fetch stage due to an instruction cache miss, even a foreground context switch can be effectively masked.

The figure illustrates that context switches happen more frequently for register files with more contexts, which is expected. However, context switch ratios are quite low in general. In fact, `NQU` and `STO` only use a small subset of registers and do not need any context switch across all three memory configurations. Also, while the overall context switch ratio increases, most context switches happen in the background. For example, most benchmarks have about the same number of foreground context switches for 2-context and 4-context configurations. As a result, most benchmarks except for `MUM` and `RAY` show negligible performance overheads for 2-context and 4-context configurations. `MUM` is the one with the highest context switch ratio. `RAY` is an outlier that we will discuss in detail later. It also appears that foreground context switches can be masked by other kinds of pipeline stalls as `BFS` and `R_BFS` do not show much slowdown even with noticeable context switch ratios.

The 8-context configuration in Figure 8(c) shows an interesting trend. There are several benchmarks, namely `LPS`, `BLK`, `RAY`, `WP`, and `R_HS`, which show significant performance degradation in the range of 10% to 30% even though their context switches are mostly categorized as background. This performance degradation comes from two sources. First, because there are only 2 warps per context in the 8-context configuration, even background switches can only partially hide the 3-cycle context switch latency. Also, these benchmarks show very high baseline IPCs in high hundreds (see



(a) 2-context register file

(b) 4-context register file

(c) 8-context register file

Figure 8: Context switch frequency (the number of context switches per instruction).

Table 2). As a result, even a small number of pipeline stalls significantly degrades the performance.

Figure 9 shows the performance overheads when the total number of threads per core is increased from 1024 to 2048. As shown in the figure, all benchmarks that previously had high performance overheads for the 8-context configuration show much lower overheads. In fact, the performance overhead is comparable to that of the 4-context case with 1024 threads per core. This shows that the number of warps per context is an important factor in performance overheads. With 1024 threads per core, the 4-context configuration has 4 warps per context, which is the same for the 8-context configuration with 2048 threads per core. Overall, with 2048 threads per core, the performance loss is 0.2%, 0.6%, and 2.2% for the 2, 4 and 8-context configurations respectively.

`RAY` shows a noticeable performance degradation although it incurs no context switches under the 2-context configuration and infrequent switches under the 4-context case. In this case, we found that the scheduling is the main source of the overhead. `RAY` only uses 12 warps per core, which results in an imbalance in the number of warps per context: 8 and 4 in the 2-context case. The greedy scheduling algorithm that

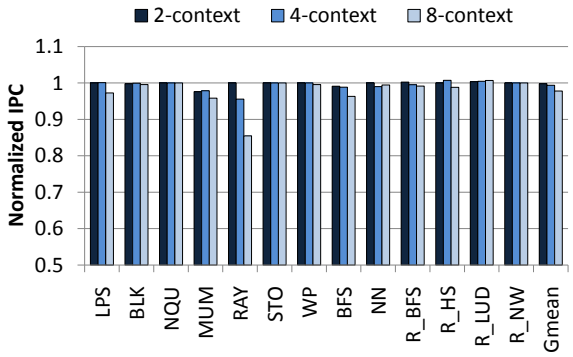


Figure 9: Performance comparisons for hybrid register files (2, 4, and 8 contexts) with 2048 threads per core, under 3 cycle context switch latency and 32 executions units per core. The performance is normalized to the baseline SRAM register file.

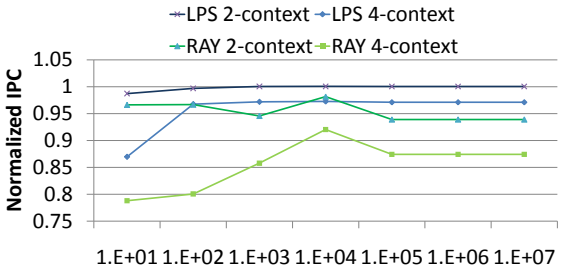


Figure 10: Scheduling trade-offs between fairness and context switches.

tries to reduce context switches tends to favor the larger context. Figure 10 shows the performance of `RAY` and `LPS` as we vary the scheduler counter threshold that determines how long one context can run without being forced to context switch. For `RAY`, the figure clearly shows the tension between fairness and context switches. Small thresholds hurt performance by increasing the number of context switches. Large thresholds also result in non-optimal performance because only a subset of warps tend to finish early. This scheduling anomaly, however, is rare and the rest of the benchmarks behave much like `LPS`, which is relatively insensitive to the scheduler counter threshold.

Although not shown here due to the space limit, we have also analyzed the performance with different context switching latencies. In general, a higher context switch latency increases performance overheads, especially for benchmarks with noticeable context switch ratios. With a higher latency, background context switches are less likely to be hidden, and the foreground context switches would cause more stalls. For the 5 cycle switching latency, the performance loss is 0.5%, 7.8% and 22.6% for the 2, 4 and 8-context configurations respectively.

We have also studied the effect of a fewer number of execution units by reducing the number of execution units per core to 16. This effectively doubles the execution latency to 2 cycles per warp instead of 1. With a longer warp execution latency, there is a larger window to perform a background context switching, which in turn improves performance. The performance improvements are especially significant in the 8-context configuration, which cannot hide a context switch

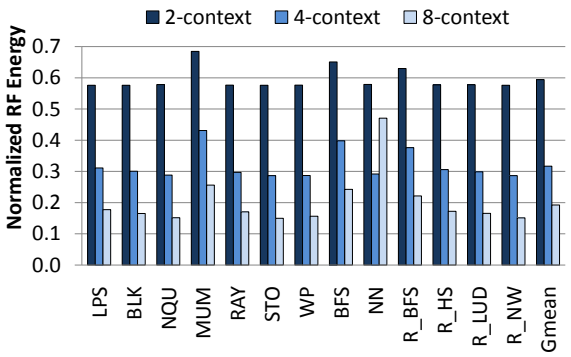


Figure 11: Normalized energy consumption for hybrid register files (2, 4, and 8 contexts). The results are normalized to the baseline SRAM register files.

in the default configuration. With the 2-cycle execution latency, the average performance loss becomes 0.1%, 1.4% and 3.2% for the 2, 4 and 8-context configurations respectively.

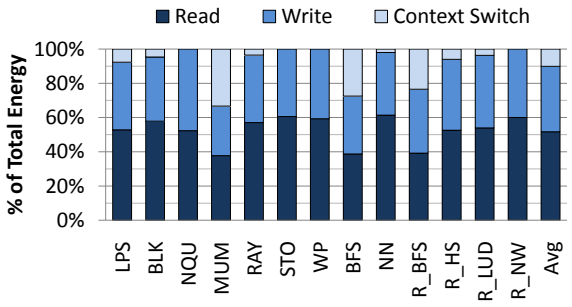
DRAM refresh is another potential source of overheads as they may incur additional context switches. However, we found that the refresh operation almost never happens in practice because contexts are scheduled frequently enough. The 2.3 ms refresh time allows a window of 1,400,000 cycles before a refresh is needed. Simulation statistics indicate that contexts are scheduled again well before this window is met. In the worst case, a context remains dormant for 52,000 cycles, and the average case was below 1,000 cycles.

5.4 Register File Energy Consumption

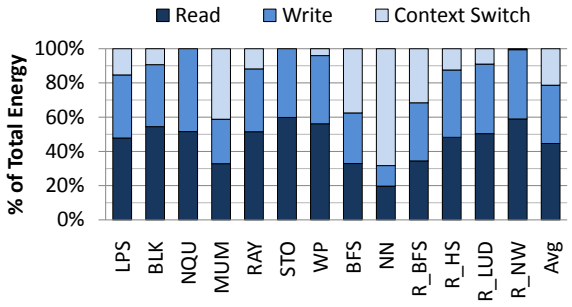
Figure 11 shows the estimates of register file energy consumption for the hybrid memory configurations. The results are normalized to the baseline SRAM array. Hybrid memory arrays show a significant reduction in total register file energy compared to the baseline SRAM array. As we increase the number of contexts for hybrid memory, the total energy continues to drop. The energy reduction can be attributed to the reduced read and write energy of the physically smaller hybrid SRAM-DRAM arrays as seen in Table 4. This result indicates that the energy savings on regular read and write operations are more significant than the additional energy consumption in context switches. Overall, the register file using 4-context hybrid memory arrays will consume roughly 68% less energy than the SRAM register file.

Figure 12 shows the energy breakdown of the hybrid memories for the 4 and 8-context cases. As we increase the number of contexts, more frequent context switching is required, and the percentage of the context switch energy in the total register file energy increases correspondingly. For `NN`, the figures show that an increase in context switches may outweigh the reduced energy in read and write operations. The energy consumption of `NN` in the 8-context configuration is higher than that of the 4-context configuration. We can also see that benchmarks with more frequent context switches, such as `MUM` and `BFS`, show less energy savings.

We also studied the leakage power in memory arrays as shown in Table 4. As the hybrid memories use less SRAM cells for the same storage, they consume only 74%, 37%, and 23% of the leakage of an SRAM array with the same capacity for 2-context, 4-context, and 8-context configurations. However, the leakage energy is not significant for the overall energy as it is orders of magnitude smaller than others.



(a) 4-context register file



(b) 8-context register file

Figure 12: Register file energy breakdown for selected benchmarks.

6. CONCLUSION

This paper presents SRAM-DRAM hybrid memory with a main application to an efficient implementation of multi-threaded register files. We laid out the proposed memory cell and arrays in an IBM 65nm process with an estimated area for an embedded DRAM cell. Thanks to the compact eDRAM cells, the hybrid memory arrays with multiple DRAM contexts provide significant area and energy savings compared to the SRAM array with the same capacity. This hybrid memory configuration is shown to be a nice match for a highly multi-threaded register files in GPUs that need to hold a large number of registers but only access a small part at a time. We modified a GPU pipeline and a scheduler for the hybrid register files. Simulations show that it is possible, with 4-context SRAM-DRAM hybrid arrays, to achieve 38% area and 68% energy savings on average with no appreciable loss (1.4%) in performance.

While we focused on reducing the area and energy of register files in this paper, we believe that the hybrid memory can be applied in many other ways and areas. For example, a hybrid memory can be used to increase register file size while remaining within fixed area and energy budgets. The hybrid memory can also provide near-instant checkpointing capabilities or improve the efficiency of other memory structures such as cache.

7. ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation grants CNS-0932069, CNS-0708788, and NSF-0646547, the Air Force Office of Scientific Research under grant FA9550-09-1-0131, and an equipment donation from Intel Corporation.

8. REFERENCES

[1] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163 –174, April 2009.

[2] J. Barth, W. Reohr, P. Parries, G. Fredeman, J. Golz, S. Schuster, H. Matick, R. Hunter, C. Tanner J Harig, H. Kim, B. Khan, J. Griesemer, R. Havreluk, K. Yanagisawa, T. Kirihata, and S. Iyer. A 500MHz random cycle 1.5ns-latency, SOI embedded DRAM macro featuring a 3T micro sense amplifier. In *Proceedings of the IEEE International Symposium on Solid-State Circuits Conference (ISSCC)*, pages 486–617, Februrary 2007.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE Workload Characterization Symposium*, pages 44–54, 2009.

[4] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 1–11, December 2010.

[5] H. M. Haynie, J. M. Turner, J. C. Hanscom, M. Cadigan, N. Hadzic, D. Di Genova, J. Aylward, S. W. Salisbury, P. Sciuto, T. D. Needham, C. E. Bubb, and R. B. Tremaine. IBM system z10 open systems adapter ethernet data router. *IBM Journal of Research and Development*, 53(1):8:1 –8:12, January 2009.

[6] D. Kanter. Inside Fermi: Nvidia's HPC Push, 2009. http://www.realworldtech.com/page.cfm?ArticleID=RWT093009110932.

[7] D. Kanter. NVIDIA's GT200: Inside a Parallel Processor, 2009. http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242.

[8] D. Kanter. AMD's Cayman GPU architecture, 2010. http://realworldtech.com/page.cfm?ArticleID=RWT121410213827.

[9] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[10] R. E. Matick and S. E. Schuster. Logic-based eDRAM: Origins and rationale for use. *IBM Journal of Research and Development*, 49(1):145–165, January 2005.

[11] U. G. Nawathe, M. Hassan, K. C. Yen, A. Kumar, A. Ramachandran, and D. Greenhill. Implementation of an 8-core, 64-thread, power-efficient SPARC server on a chip. *IEEE Journal of Solid-State Circuits*, 43(1), January 2008.

[12] NVIDIA Corporation. NVIDIA CUDA Programming Guide, 2.0 edition, 2008.

[13] D. A. Patterson and C. H. Sequin. RISC I: A reduced instruction set VLSI computer. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 443–457, 1981.

[14] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4):505–521, July 2005.

[15] J. Stokes. Microsoft beats Intel, AMD to market with CPU/GPU combo chip, 2009. http://arstechnica.com/gaming/news/2010/08/microsoft-beats-intel-amd-to-market-with-cpugpu-combo-chip.ars.

[16] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, January 2002.

[17] TSMC. Embedded memory, 2010. http://www.tsmc.com/download/brochures/2010_Embedded_Memory.pdf.

[18] UMC. Embedded memory, 2011. http://www.umc.com/english/pdf/eMemory.pdf.

[19] A. Valero, J. Sahuquillo, S. Petit, V. Lorente, R. Canal, P. López, and J. Duato. An hybrid eDRAM/SRAM macrocell to implement first-level data caches. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 213–221, 2009.

[20] D. Wendel, R. Kalla, R. Cargoni, J. Clables, J. Friedrich, R. Frech, J. Kahle, B. Sinharoy, W. Starke, S. Taylor, S. Weitzel, S. Chu, S. Islam, and V. Zyuban. The implementation of POWER7^{TM}: A highly parallel and scalable multi-core high-end server processor. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, pages 102 –103, February 2010.